The H-Machine

Experiments in the Organic Growth of Interacting Data Structures

Lawrence A. Stabile

Abstract—**The H-Machine is a hypergraph-based language and interpreter, based on rule matching by hypergraph isomorphism. Rules are part of the hypergraph being constructed, and can be matched and modified as can other data, thus forming a complete meta-system. The H-Machine language is extremely simple, yet can express a wide range of data structures and computations, with inherent parallelism. The matching system allows a full range of expression of recursive relations, implicit iteration, and, via meta-rules, a modular way to describe computation at a high level.**

Data and rules have a simple graphical interpretation which offers good visibility into the resulting structures. Examples illustrate both data and rule structures, derived data flow graphs, cellular automaton matrices, ancillary relations, and rules – predicate subgraph, edges to be added, edges to be deleted -- within a single graphical model. The Graphviz toolkit is used for rendering. Implementation is briefly described, with links to code and a gallery of generated H-Machine runs.

Keywords—FFT, hypergraph, meta-system, Rule-30, rule-based-system.

I. Introduction

As computing has become ever more pervasive, and ever larger in scale, parallel processing techniques have become essential tools. Cellular and graph automata have long been exemplars of systems with a high degree of natural parallelism, as have rule-driven systems, such as blackboards, that operate as independent agents on a set of shared data.

The H-Machine (H for "hypergraph") is a simple graph automaton, driven by a rule language called simply H. Rules are incorporated into the graph and thus form a complete meta-system. With rules so incorporated, the system can control its own behavior. An analogous model is that of a blackboard system whose rules themselves are on the blackboard, and able to be manipulated and controlled within the system. Such systems have been explored, for instance, by Davis[4] and Davis and Lenat [5].

The H-Machine operates analogously to an organic or molecular system. Objects, i.e. subgraphs, must be brought together for interaction in some way, just as molecules must interact for chemical reactions to occur. The interaction in the H-Machine is driven by isomorphism among the subgraphs.

II. The Pure H-Machine

A hypergraph is a generalization of a graph wherein an edge can consist of more than two nodes. A hypergraph *h* then is a collection of subsets of some universe *U*. A subgraph of a hypergraph *h* is simply a subset of *h*. See for example Berge [2] for a treatment of hypergraph theory.

In the H-Machine, a hypergraph G is established which grows according to rules. Rules are themselves subgraphs of G, and have a predicate part and a consequent part. The predicate is matched by isomorphism against other subgraphs of G, and the mappings of the pattern to the data nodes and edges are used to instantiate new edges defined in the consequent.

Hypergraphs are useful due to their generality, offering a wide variety of data structures in a uniform manner. A pure form of the H-Machine starts with an initial graph G_0, which then evolves as follows:

$$G_n = G_{n-1} \cup \bigcup_{i,j} r\left(subgraphs\ (G_{n-1})_i, (subgraphs\ (G_{n-1})_j\right)$$

where

$$r(h_r, h_d) = \phi \text{ if } h_r \text{ is not a rule or}$$
$$h_r.pred \text{ is not isomorphic to } h_d;$$
$$\text{else a graph, the new edges,}$$

and

$$subgraphs\ (h) \equiv \text{collection of all subgraphs of } h.$$

The function *r* detects whether h_r is a rule subgraph, and if so, extracts the predicate (*pred*) and the consequent (*add*) subgraph components. If an isomorphism exists between $h_r.pred$ and h_d, then there exists a mapping $f: nodes(h_r.pred) \rightarrow nodes(h_d)$. A node $n \in nodes(h_r.add)$ is replaced by $f(n)$ for each edge in $h_r.add$, thus instantiating new edges.

III. Building a Practical Version

A straightforward evaluation of the above formula for G_n implies finding all subgraphs of G_{n-1} and matching each rule subgraph against each other subgraph (rule and non-rule). Clearly this is intractable for any but the smallest graphs.

A practical model for the H-Machine is inspired by basic physical ideas. The set of nodes is similar to atoms, subgraphs to molecules. Rules are analogous to special molecules which induce reactions when they interact and match with other

molecules. Reactions involve creating new edges and forming new subgraphs.

Rather than an exhaustive model where we try each subgraph, rules are attached to nodes and a node may be *executed*, wherein the set of attached rules is matched against subgraphs in the neighborhood of a node. This results in a controlled spreading-activation model. Rules to be matched need to be "near" each other, just as molecules need to be physically close to each other to react.[1] Because rules are subgraphs within G, control is provided by passing, copying, and modifying rules, via other rules. [2]

A rule may match a subgraph in more than one way, in which case each such a match is used to produce new edges. This eliminates the need for an explicit notation for iteration.

A means is introduced to create new nodes, as part of the rule predicate, using distinguished attribute and node names. This is required for any practical form of a graph model Deletion of edges is also introduced as a practical matter, although its use has been kept reasonably low.

IV. The H Language

The H language has a simple syntactical form, and a simple graphical form. In this brief paper, we will describe these largely by example. See [10] for more extensive descriptions of the H-Machine and the H language.

Lisp notation and primitive datatypes are used. An edge is an ordered n-tuple, denoted as a list. A node is a string, number, or symbol. Thus

```
   (x next y) (n235 value 5)(n356 red)
(book42 name "Huckleberry Finn")
```

are all edges. Note that while the Pure H-Machine is defined using unordered sets, the use of n-tuples is much more efficient in implementation, and causes no loss of expressiveness.

The graph G being built is thus simply a set of lists. Nodes can be symbols, numbers, or strings. Nodes and edges are unique; i.e., there is no duplication in G. G is connected, and all rules are contained therein. There are no "hidden rules" in the kernel.

A rule consists of a predicate (pred) part, and a consequent (add) part. The pred is matched against a subgraph found in a neighborhood of the node to which the rule is attached. Matching is by hypergraph isomorphism, i.e., a mapping is sought between the nodes of a rule *r* and the nodes of a subgraph of G in the neighborhood of the node *n* to which *r* is attached, such that edge relationships are preserved.

For example, to define the transitivity of <, we write:

```
(rule
 (name trans-less-than)
 (pred
  (?x < ?y)
  (?y < ?z))
 (add
  (?x < ?z)))
```

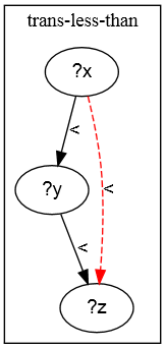
When this rule is run it will match the subgraph

```
(3 < 4)(4 < 5)
```

and produce

```
(3 < 5)
```

The graphical form of this rule is:



The dark, solid nodes and edges represent a graph which must match a subgraph in G. Upon such a match, mapped nodes in G are bound to the indicated variable nodes. The red dotted edges are then produced, based on those bindings.

This example shows the convention that three-node edges are interpreted as object-attribute-value pairs. Two-node edges by default are drawn as a named node, an unlabeled edge, and an unboxed property value. Four-node edges having the third node marked as a *two-input-op* are taken to be of the form (input1 input2 op output) and drawn as a two-input, one-output dataflow function node. See Fig. 1 for examples of these forms.

One iterates over a set by using the multi-match properties of the H-Machine. For example

```
(rule
 (name mark-done)
 (pred
  (?opset type operations)
  (?opset elem ?op))
 (add
  (?op done)))
```

will add the *done* marker to each element ?op, a member of the set ?opset.

New nodes are created using the distinguished relation *new-node*. However in the spirit of the Platonic view that nodes all really exist already, we denote new nodes in the *predicate* (pred) part of the rule. For example

```
(rule
 (name add-link)
 (pred
  (?x next ?y)
  (?n new-node sn1))
 (add
  (?y next ?n)))
```

This says in essence that if you match against this new-node edge, you get a new node, bound to ?n. Plato springs to life!

[1] For development purposes, we can short-cut this locality using a global form of rule, which serves as a way to "wash" rules over a set of nodes, looking for matches. This is generally inefficient but allows for development and later optimization by localizing the rules and passing them as required.

[2] Rule copying is accomplished within the language and has some interesting properties. Omitted from this paper due to space considerations, please see [10] for an explication.

The sn<number> syntax is special, and causes (eventually) the creation of an actual new node (n<number>), which can then be used in the add clause.[3]

A rule can specify any number of new-node relations, the sn<number> designations are considered scoped within their defined rule [4]. The variable to which the new node is bound is arbitrary (?n in the above example).

V. Extended Example: The FFT Butterfly

The Fast Fourier Transform (FFT) may be written as

$$F_N(x) = G\,(F_{N/2}\,(\text{odd}(x), F_{N/2}\,(\text{even}(x))) \qquad (1)$$
$$F_1(x) = x,$$

where x is an array of length N, and G is a linear combination based on the complex exponential W_N [13]. The algorithm takes a classic divide-and-conquer approach to a problem that is "naturally" quadratic and converts it to $O(nlogn)$ or closely similar complexity. Writing it using direct top-down recursion retains the $O(nlogn)$ behavior, but adds a considerable extra factor of overhead, so FFTs in the real world are normally coded bottom–up as tight loops, and will employ such tools as parallel threads, hardware dataflow, and a host of other techniques.[8]

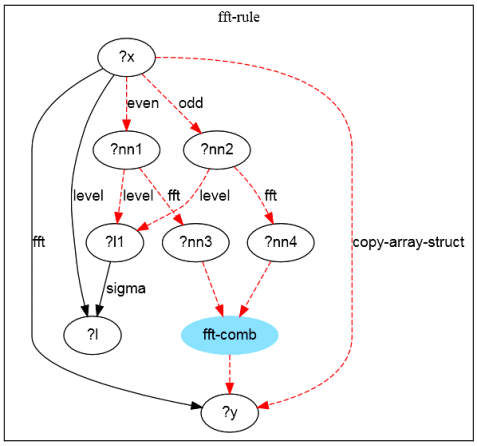
The H-Machine version of the FFT connects the top-down and bottom-up worlds, by deriving the bottom-up butterfly model directly from the recursive equations, via rule-driven transformations. Note that in this experiment only the butterfly data flow topology is formed; actual numeric calculation is not made. In the description to follow, please see the rules in h.lisp [10].

We assume that G is initially populated with a set of rules, and with a *sigma* chain, representing the natural numbers, out to some reasonable but small cardinality. We start by defining a node to be the top of a tree of a certain number *N* of levels. Rules match this and add two nodes downward for each existing node, stopping when the level reaches zero. Other rules produce the *tree-next* relation, which connects nodes across the tree. The result of this propagation is that all the bottom-level nodes will be connected together by *next*. The tree is asymmetrical, in that the initial node defines that the two next-to-top nodes will have the *zero* and *max* property (the choice of which node has which property is arbitrary). Rules propagate these properties down the tree, and when the bottom is reached, a loop rule detects the *zero* and the *max* at the bottom level, and then connects them together, forming the *next* loop, with a distinguished *zero* node. The *next* chain is of length 2^N. Forming the chain into a cycle is needed to emulate the *mod* operation required to construct the butterflies.

With an array thus defined, we need to take the odd and even parts. First, rules running on each array mark nodes as even (*ev*) or odd (*od*), with the basis that zero is even. Then other rules construct a new array of only the points marked odd or even; they are then endowed with loops and a distinguished zero node, as above.

Rules for FFT processing detect these arrays, and connect them together via the combining function *fft-comb*. The set of these now forms the basic bottom-up flow graph. More rules detect these combining functions, and form the butterflies with the array elements, as a set of half-butterflies (*fft-hb*).

The preceding is defined at the top level by the fft-rule, which detects the presence of edges of the form (*x* fft *y*)(*x* level *l*), where $l = log_2(N)$, *N* the number of points in the FFT. Fft-rule, in diagrammatic and textual form, is shown below. It is evident that the structure of fft-rule follows closely that of the recursive equation (1).




```
(rule
 (name fft-rule)
 (pred
  (?x fft ?y)
  (?x level ?l)
  (?l1 sigma ?l)
  (?nn1 new-node sn1)
  (?nn2 new-node sn2)
  (?nn3 new-node sn3)
  (?nn4 new-node sn4))
 (add
  (?x even ?nn1)
  (?x odd ?nn2)
  (?x copy-array-struct ?y)
  (?nn1 fft ?nn3)
  (?nn2 fft ?nn4)
  (?nn3 ?nn4 fft-comb ?y)
  (?nn1 level ?l1)
  (?nn2 level ?l1)))
```

In the figure above, nodes whose names are prefixed by "?" are variables, and those variables of the form ?nn<n> represent new nodes. The relation of these to the *new-node* relation is not shown, for clarity; however, these are shown in the textual form. The black solid edges are predicate edges. These must match a subgraph in G, in which case the variables are bound to the associated nodes (new nodes are included in the predicate binding). Using these bindings, the red dotted edges are then instantiated upon match.

As shown, the fft relation between two nodes (the edge (?x fft ?y) in this case) spawns more edges, some of which have fft edges. This then causes the rule to apply again, and thus the recursion is produced. Each fft is connected to a node with an

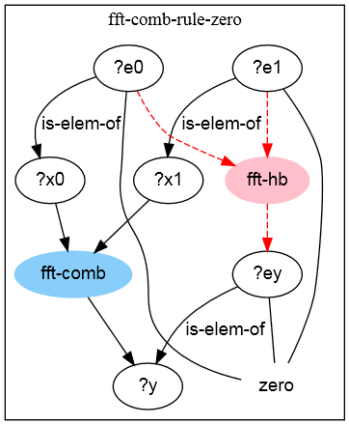
[3] It's eventually because in fact, a node of the form nn<number> is created by the rule definer, and it in turn causes a new node to be created at run time.

[4] The naming convention derives from "scoped".

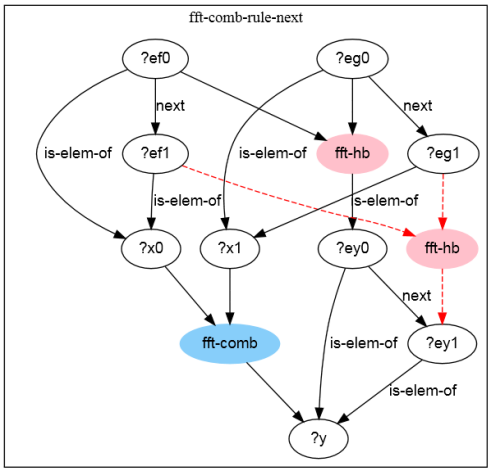
[8] See [1] and [3] for surveys of current FFT architectures.

odd or *even* relation. As described above, those relations are also detected, and spawn more rules to produce the odd and even arrays. The crucial edge copy-array-struct is added, which causes the array structure of the input node to be copied to the output node. This is needed to form the full set of nodes required for the butterflies. Finally, the FFTs of the reduced odd-even relations are combined using the *fft-comb* node. Note in the H-Machine this is simply a four-node edge of the form (<input1> <input2> fft-comb <output>). Fft-comb is tagged with the property *two-input-op* and this is recognized by the graphics dumper and turned into a standard dataflow function graphic.

Many other rules must run to produce the full FFT butterfly graph. Key is the fft-comb-rule set of rules, which detect the *fft-comb* and produce the butterflies from the array elements it references. The basis rule for this is fft-comb-rule-zero (below), which detects the *zero* elements connected by an *fft-comb* and connects those elements via a half-butterfly (*hb*).



Fft-comb-rule-next, below, then detects that a half butterfly has been contstructed and builds one attached to the next element, and so forth down the line. These rules then fill in all the butterflies at all levels.



The results of this are shown in the following series of figures, for an 8-point FFT between the nodes x and xfft. shows the recursive top-down/bottom-up structure using relations that display the symmetry and flow well. The relation *d* was defined solely for the display as a convenient "connection" between the top-down and bottom-up parts of the graph growth process. There is no actual "data flow" to be perceived across the *d* relation. Fig. 2 shows the construction of the butterflies.

The rules used to generate the FFT structures, and a combined form of Fig. 1 and Fig. 2 is shown in Fig. 3. See also the rule text, and the gallery for higher-resolution graphics and the complete set of rules [10].

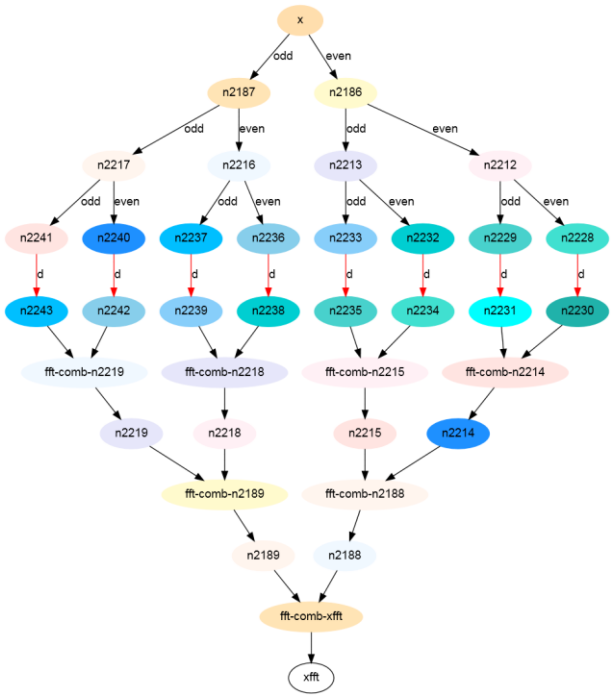


Fig. 1. FFT, showing symmetry of top-down and bottom-up components

Fig. 2. FFT, the generated butterflies

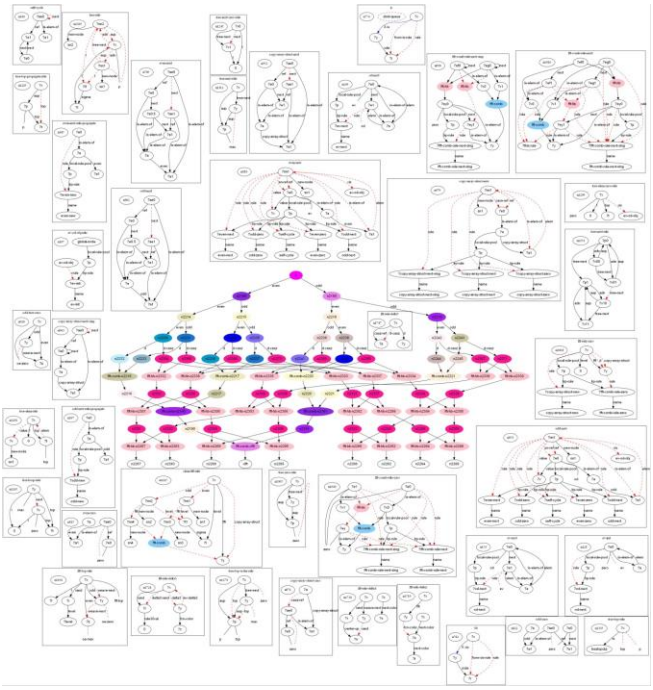


Fig. 3. 8-point FFT, complete top-down/bottom-up construction, with the essential rules which generated it. In [10], see gallery/fft-8-just-fft-rules.svg for a higher-resolution version.

VI. Graph Mutation

We will continue to use the FFT example as part of describing mutation rules and structures. *Mutation* is defined as changes made to some graph structure originating from rules which are outside the rules used to create the structure. For example, generating a tree using the tree rules, we might define a rule which operates on a node so many levels down from the top and so many across, and adding onto or modifying the surrounding structure in some way. However instead of such literal-address types of rules, we seek more general ways to specify how a graph should mutate as it grows.

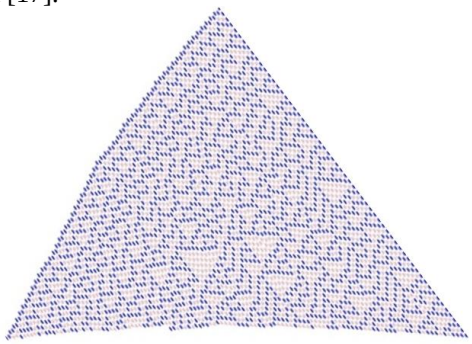
One way to do this, described in this section, is to develop multiple kinds of structures, and let them interact in some way. We'll use three of them: one of them is a tree, as described above.

The second is based on the Rule-30 one-dimensional cellular automaton from [17]. This is built as a set of unfolding layers, each of which adds a new row of nodes in accordance with the Rule-30 rules.

The primary reason for choosing Rule-30 is that it provides a pseudo-randomness capability, i.e., down the center cell. [9] Secondarily, it provides good exposition for rule-generating rules. This is covered in a later section, Meta-Rules .

To build a rule-30 graph, a set of layers is built based on an initial natural number parameter. A rule-30 node has relations *up*, *next*, and *rule-30-val*, the latter of which is 1 or 0. Note these values show in the graphics as nodes colored blue and pink, respectively. *Up* refers to the node at the previous layer, and *next* refers to the adjacent node in the layer. A rule for each pattern of interest checks the *up* and *next* nodes for the matching pattern of *rule-30-vals*, and creates new nodes with appropriate values for the next layer. Meanwhile another rule tracks the *up* relation along the center node, and propagates a *center-up* relation down the center. When the bottom is reached, a loop rule joins the bottom and top via *center-up*, and this is used subsequently as a circular pseudo-random-value generator.

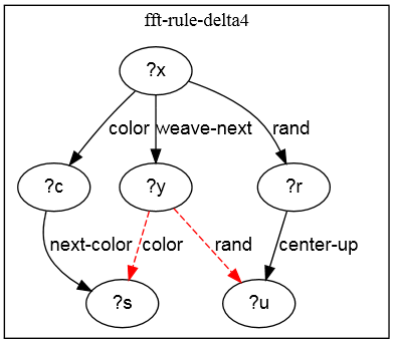
The figure below shows a rule-30 run carried out to 60 levels. [10] Note the distinctive triangular pattern seen in the figure, as in [17].



The third interacting structure is the *color circle*, a ring of colors related by the *next-color* relation. This is built from literal data. The names of the colors are selected from those supported by Graphviz [8].

An additional useful structure to build onto a tree is a *linear weave*, a chain through all the nodes in the tree. This is added to the odd-even tree via the weave-next and related rules. This supplies a linear ordering to the tree nodes. The rules for this can be found in h.lisp [10].

The rule fft-rule-delta4, shown below, propagates *rand* and *color* along the weave chain, looping back for color or rule-30 node should it reach the end of its initial chain (the *next-color* relation for colors and the *center-up* relation for rule-30).



Thus, fft-rule-delta4 assigns a color to each node, which produces the colorization noted in the figures. Fft-rule-delta4 also supplies each node with a pseudo-random bit, given by the relation chain *rand.rule30val*.

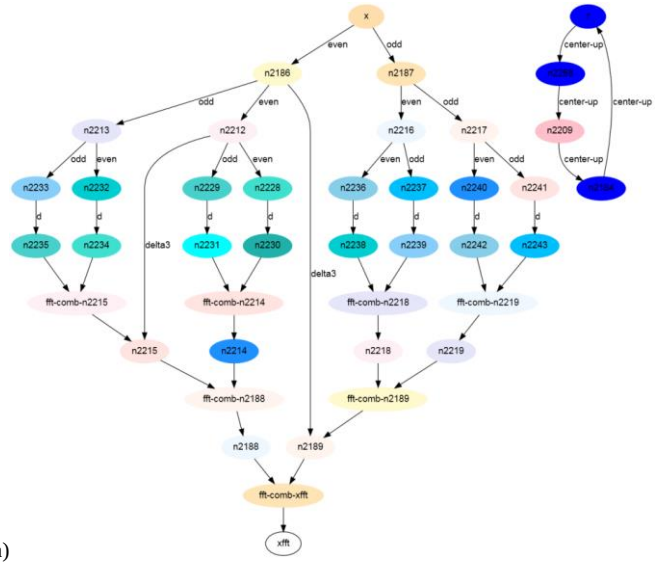
[9] See [17] and [7] for analyses of the randomness properties of Rule-30.

[10] Note the last (bottom) row is incomplete.

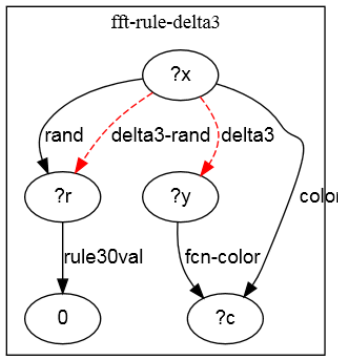
Given then the assignment of random bits and colors to the nodes, the rule fft-rule-delta3 ((b)). says that if for two nodes *x* and *y*, x.*color* = *y.fcn-color*, and *x.rand.rule30val* = *0*, then we'll add a new edge, *delta3*, between *x* and *y*. The relation *fcn-color* is attached to an fft-comb output node and indicates the color of the associated *two-input-op* which impinges on *y*.

We are thus able to combine conjunctively the selection properties of these structures, rule-30 and the color-circle, using a simple compact rule.

The result of this is shown in Fig. 4 (a). This is an 8-point FFT, but the butterflies are not shown since they are not affected by the mutation. Two edges are chosen for addition based on the criteria, each marked *delta3*. Note that the color of node n2186 is the same as that of fft-comb-n2189, and that of n2212 is the same as fft-comb-n2215. The upper-right part of the figure shows the *center-u*p random sequence, in this case for a rule-30 iteration size of only four.



(a)



(b)

Fig. 4. Delta3 graph mutations (a), and corresponding rule fft-rule-delta3 (b).

Note that fft-rule-delta3 also installs an edge, *delta3-rand*, from the source node to the rule-30 node which matched. This provides a view into the random part of the mapping which assigned the delta3 edges. Displaying this mapping gives us an interesting looking view, shown in Fig. 5, of rule 30 "eating" the FFT.

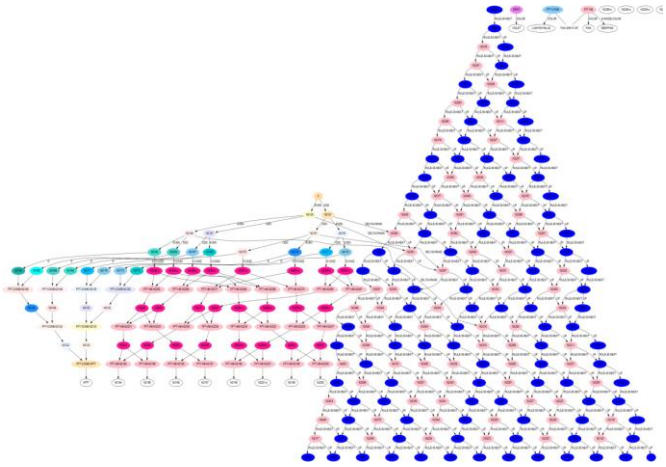


Fig. 5. Rule 30 eats the Fourier Transform

VII. Meta-Rules

In the H-Machine, rules can match on and manipulate other rules. This is accomplished by matching on the edges that make up the components of a rule (primarily its predicate and consequent clauses), and generating edges which represent components of new or existing rules. The representation is such that it's possible to manipulate rules solely with the edge-manipulation syntax described earlier. However, this can be unwieldy, so syntactic sugar is added to ease the complexity.

Rules are represented within a graph G by expanding the edge lists in pred, add, and the other components, into nodes that explicitly define the edge content as relations in G. For example, the syntactical form

```
(rule
  (name rule1)
  (pred
     (?x next ?y))
  (add
     (?y prev ?x)))
```

is transformed by the kernel to the set of edges

```
(r name rule1)
(r pred p1)
(p1 elem0 ?x)
(p1 elem1 next)
(p1 elem2 ?y)
(r add a1)
(a1 elem0 ?y)
(a1 elem1 prev)
(a1 elem2 ?x)
```

where r, p1, and a1 are new nodes created by the kernel.

By matching on these edges we may create or modify rules. To make the syntax a bit more friendly, two pieces of syntactical sugar are added, (1) nested rules and (2) rule component lists.

1) Nested Rules

Nested rules allow a rule to be created as a new node, the element of any added edge. The values of variables bound in the creating rule are used in the new rule, and coupled with the non-deterministic aspects of matching allow us to generate a family of rules from a concise rule-generation description.

An example of this is shown below, the rule generator for the Rule-30 cellular automaton.

```
(rule
 (name rule-30-rule-gen)
 (pred
  (rule-30-data ?xval ?yval ?zval ?nval))
 (add
  (global-rule-pool-node grp-rule
    (rule
     (pred
      (?y level ?l)
      (?l1 sigma ?l)
      (?x next ?y)
      (?y next ?z)
      (?y interior)
      (?x rule30val ?xval)
      (?y rule30val ?yval)
      (?z rule30val ?zval)
      (?nn1 new-node sn1))
     (add
      (?nn1 up ?y)
      (?nn1 interior)
      (?nn1 level ?l1)
      (?nn1 rule30val ?nval)))))
 (del
  (global-node rule ?this-rule)))
```

The nested rule descriptor is recognized at rule definition time and converted to the set of edge representations needed to specify the new rule. Note that binding of nodes to be used as part of the new rule occurs at runtime, i.e., when the nested rule is actually created. Thus nested rules as written syntactically supply a lexical scope, but this is just shorthand -- the resolution of bindings occurs completely at runtime.

The variables ?xval, ?yval, ?zval, and ?nval represent the left, center, and right cells of a triple, with the new value, under the center, denoted by ?nval. These variables need to match against some values that denote the desired cell pattern. This is done by the Rule-30 data generator:

```
(rule
 (name rule-30-data)
 (pred
  (global-node rule ?rule-30-data)
  (?rule-30-data name rule-30-data))
 (add
  (global-node rule-30-data)
  (rule-30-data 0 0 0 0)
  (rule-30-data 0 0 1 1)
  (rule-30-data 0 1 0 1)
  (rule-30-data 0 1 1 1)
  (rule-30-data 1 0 0 1)
  (rule-30-data 1 0 1 0)
  (rule-30-data 1 1 0 0)
  (rule-30-data 1 1 1 0))
 (del
  (global-node rule ?this-rule)))
```

Each entry in the binary-matrix pattern above matches a cell pattern of rule 30. One simply changes that to get another CA rule. The rule runs on the global node and deletes itself once run.

Note that this method very compactly expresses the CA generation rules and how to carry them out. In particular, there is no explicit expression of iteration.

2) Rule Component Lists

Simplification of the specification of rule component lists is especially useful for modifying existing rules. For example, the "clean" fft-rule above will run, since it's global, but not very efficiently. From completely outside that rule, we can add another rule-modifying rule to optimize it. Shown below, fft-rule-opt detects fft-rule by matching on its name in the global pool, then adds propagators for rules that will be needed by subsequent nodes. It also adds fft-rule to the local rule pool, and removes it from the global rule pool. We thus produce a much more efficient rule, via a simple, semantics-preserving transformation.

Note the passing of variables such as ?nn1 to the modified rule. Since the variables are not bound in the predicate, they are provided literally to the target rule, which is as desired.

Since these rules run for initialization purposes, they delete themselves via the del clause, and the built-in variable ?this-rule, bound to the node of the currently running rule.

```
(rule
 (name fft-rule-opt)
 (pred
  (global-node global-rule-pool-ref
       global-rule-pool-node)
  (global-rule-pool-node grp-rule ?fft-rule)
  (?fft-rule name fft-rule))
 (add
  (local-rule-pool-node lrp-rule ?fft-rule)
  (?fft-rule pred (?x local-rule-pool ?p))
  (?fft-rule pred (?p lrp-rule
                     ?fft-comb-rule-zero))
  (?fft-rule pred (?fft-comb-rule-zero name
                          fft-comb-rule-zero))
  ...more rules and rule names...
  (?fft-rule add (?nn1 rule ?fft-rule))
  (?fft-rule add (?nn2 rule ?fft-rule))
  (?fft-rule add (?y rule
                    ?fft-comb-rule-zero))
  ...more adds...)
 (del
  (global-rule-pool-node grp-rule ?fft-rule)
  (?this-obj rule ?this-rule)))
```

VIII. Comments on Modularity

Using the H-machine for a while as a programming language, there are some interesting notions of modularity which have emerged.

First is the observation that the use of isomorphism/matching allows computation in what are effectively scoped contexts, without explicitly defining a scoping model. For instance we can use the attribute names *elem* or *next* in many objects and they do not conflict, as long as use is narrowed down as appropriate, i.e., by using other attributes to isolate the unique set of subgraphs intended to match. Sometimes one can assign a unique "type name" or similar explicit tag to refine these attributes, i.e., to create the explicit notion of an object that exists in some sense independent of its properties, but it's not required.

A side benefit is that one can match on these attributes as well, since they are nodes in the hypergraph G, and find if desired, for example, all edges which contain a given attribute.

The second notion of modularity that has emerged is similar to that found in Aspect Oriented Programming [11]. In the H-machine, rules are distinct entities and do not need to be included in some lexical scope. This property, coupled with the use of meta-rules, allows the alteration system behavior across a wide range using rules and meta-rules expressed outside the modules affected, in the same way that expressing an Aspect

can affect code semantics over a wide set of objects and/or methods. This was illustrated above in the fft-rule optimizer (see fft-delta.lisp in [10] for the complete code).

IX. Conclusion

Graph automata and rule-based systems have been well-studied, from various perspectives.[17] The primary contribution of this paper is to demonstrate the use of rules and meta-rules in a hypergraph context, using practical examples as guides. A secondary contribution rests in the simplicity of the rule system, and the versatility obtained by thinking in hypergraph terms while retaining that simplicity. The generated graphics are a relatively straightforward result of this simplicity, and provide an interesting pedagogical and developmental tool.

This paper has not emphasized implementation and performance, and the topics deserve separate treatment. The H-Machine code right now is very compact; however, it is not very efficient. Improving the efficiency can take several forms, e.g., subgraph search heuristics, indexing, rule compilation, bootstrapping, parallel processing, and a host of other possible techniques. See [15] and [16] as examples of recent work in large-graph search. Deeper study and incorporation of techniques such as these is called for in expanding the scale of the H-Machine.

Using the meta-rule system to modify and optimize other rules is another promising direction. The current H-Machine has just scratched the surface in this respect, and automation of these techniques is part of future efforts.

The ability to build bottom-up dataflow graphs directly from high-level recursive equations inspires ideas of compilation to various representations, including those of sequential and parallel machines. This appears to be another path worth pursuing.

Declarative systems such as this and others found in the literature have long held promise as a way, in particular, of separating data representations from matters of control. The H-Machine kernel provides a primitive matcher, and a complete but rudimentary control system. The meta-rules show promise in providing a way to describe control and optimization modularly and succinctly, within the scope of the language definition. It is hoped that the results in this paper help inspire further work in this area.

[17] The literature on these topics is vast and has a long history. Graph transformation is covered comprehensively by Ehrig [6]. Davis [4], and Davis and Lenat [5] did early work in meta-rules. Hewitt *et. al.*'s Omega [9] inspired the pursuit of self-contained meta-systems. Wolfram's NKS [17] provides a comprehensive view of simple CAs, and extends the ideas to graphs as well, with regard to very simple physical models and models of universal computation. Miller and Fredkin [12] describe a 3D reversible CA, also with conjectured universality. Wu and Rosenfeld [1] describe early forms of graph automata and their mathematical properties. Also similar to the work herein is Radul and Sussman's propagator model [14]. These are a small sample of the work in these fields.

X. References

[1] T.Angeline and D.Narain Ponraj. A Survey on FFT Processors, International Journal of Scientific & Engineering Research Volume 4, Issue 3, March 2013. http://www.ijser.org/researchpaper/A-Survey-on-FFT-Processors.pdf.

[2] Claude Berge. Hypergraphs, North-Holland, 1989.

[3] Anwar Bhasha Pattan, Dr. M. Madhavi Latha. Fast Fourier Transform Architectures: A Survey and State of the Art, International Journal of Electronics & Communication Technology, 2014. http://www.iject.org/vol5.4/1/25-Anwar-Bhasha-Pattan.pdf.

[4] Randall Davis. Meta-Rules: Reasoning about Control, MIT AI Lab Memo 576, March 1980.

[5] Randall Davis and Douglas B. Lenat. Knowledge-Based Systems in Artificial Intelligence, McGraw-Hill, 1982.

[6] Hartmut Ehrig *et. al.* Fundamentals of Algebraic Graph Transformation, Springer, 2006.

[7] Dustin Gage, et. al. Cellular Automata: Is Rule 30 Random?, http://www.cs.indiana.edu/~dgerman/2005midwestNKSconference/dgelbm.pdf

[8] Graphviz, http://www.graphviz.org.

[9] Carl Hewitt, Giuseppe Attardi, Maria Simi. Knowledge Embedding in the Description System Omega, Proceedings of the First AAAI Conference on Artificial Intelligence, 1980.

[10] H-Machine code and pictures, https://app.sugarsync.com/wf/D744968_07071716_9998333. Most easily accessed by downloading the zip file (about 26 MB). Note that most of the graph dumps contain the rules as well.

[11] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J. M.; Irwin, J. Aspect-oriented programming, Proceedings of the 11th European Conference on Object-Oriented Programming, 1997, http://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf

[12] Daniel B. Miller and Edward Fredkin. Two-state, Reversible, Universal Cellular Automata In Three Dimensions, 2005, http://www.digitalphilosophy.org.

[13] Alan Oppenheim and Ronald Schafer. Digital Signal Processing, Prentice-Hall, 1975, page 290.

[14] Alexey Radul and Gerald Jay Sussman. The Art of the Propagator, MIT-CSAIL-TR-2009-002, 2009.

[15] Xuguang Ren, Junhu Wang. Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs, Proceedings of the VLDB Endowment, Vol. 8, No. 5, 2015

[16] Zhao Sun, *et. al*. Efficient Subgraph Matching on Billion Node Graphs, Proceedings of the VLDB Endowment, Vol. 5, No. 9, 2012.

[17] Stephen Wolfram, A New Kind of Science, Wolfram Media, 2002.

[18] Angela Wu and Azriel Rosenfeld, Cellular Graph Automata. I. Basic Concepts, Graph Property Measurement, Closure Properties, Information and Control 42, 305-329, 1979.